# A metadata action language

Keith Golden
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
kgolden@ptolemy.arc.nasa.gov

May 3, 2001

**Abstract**

The data management problem comprises data processing and data tracking. Data processing is the creation of new data based on existing data sources. Data tracking consists of storing metadata descriptions of available data.

This paper addresses the data management problem by casting it as an AI planning problem. Actions are data-processing commands, plans are dataflow programs and goals are metadata descriptions of desired data products. Data manipulation is simply plan generation and execution, and a key component of data tracking is inferring the effects of an observed plan. We introduce a new action language for data management domains, called ADILM. We discuss the connection between data processing and information integration and show how a language for the latter must be modified to support the former. The paper also discusses information gathering within a data-processing framework, and show how ADLIM metadata expressions are a generalization of Local Completeness.

# 1 Introduction

## 1.1 The NASA data management problem

A big part of NASA's job is data management. Satellites, unmanned spacecraft, planetary rovers and observatories, for all their complexity, can all be viewed as remote sensors; their sole purpose is to gather data, which are then processed, delivered to the scientific community and the public, and archived. The data management problem is especially acute in the Earth Sciences, where the data sets are large and diverse, and there is an increasing demand for real-time data processing in support of a wide range of scientific tasks (cite SensorWeb). Real-time data processing in support of novel science goals is not feasible with the current technology, since a substantial amount of work is involved in acquiring the data and converting them to the appropriate formats. These activities can be scripted, but the scripts themselves take time to write and debug. Also, there is a tremendous problem in *data tracking*: keeping track of what data exist, what the data represent, and where they are stored.

Figure 1 shows a typical data processing operation. Sets of images that were taken by a black and white camera, through red, green and blue filters, are first combined into color composite images. These images, each representing a narrow field of view, are then combined into a mosaic, corresponding to a larger field of view. The mosaic is then compressed and put on the web. Arcs in the figure represent dataflow. Data processing consists of constructing and executing such dataflow programs to produce a desired result. Data tracking consists of deriving a metadata description of data products produced by such programs, and storing it in a database, to facilitate later data searches.

We address the data management problem by casting it as an AI planning problem. Actions are data-processing commands, plans are dataflow programs like Figure 1, and goals are metadata descriptions of desired data products. Data manipulation is simply plan generation and execution, and a key component of data tracking is inferring the effects of an observed plan. We introduce a new action language for data management domains, called ADILM.

Although data tracking is the easier problem computationally, it places more demands on the representation of data and the actions that manipulate data, since the system must be able to generate correct and usable metadata descriptions for any given dataflow program, regardless of how it was generated. For example, Collage [?] and MVP [?] both use planning to automate data manipulation (in particular, image processing), but they do not automate data tracking, and in fact neither uses a representation that is suitable for metadata generation. Both use an HTN representation, which allows them to avoid providing a detailed causal theory of data processing. However, it is precisely this detailed causal theory that is needed to determine the effects of arbitrary data-processing plans.

## 1.2 Roadmap

The remainder of the paper is organized as follows. Section 2 discusses the connection between data processing and information integration. Section 3 discusses how data-producing actions are represented to facilitate reasoning about data-processing plans. Section 4 discusses the representation of metadata. Section 5 discusses the representation of "filter"

Figure 1: A dataflow plan. First, separate monochrome images taken through red, green and blue filters are combined to form a color image. Then, these images are tiled to form a mosaic. Finally, the full resolution image is archived and a JPEG-compressed version is stored on a public website.

actions that transform data. Section 6 discusses the representation of actions that deliver data to a destination. Section ?? discusses the representation of dataflow plans. Section 8 discusses how to reason about these plans. Section 7 discusses how to gather information in a data-manipulation paradigm.

# 2 Data manipulation vs Information Integration

There has been substantial work in the area of planning for information integration. This work can be broadly characterized as extracting information from a number of data sources and combining the results in order to answer a user query. For example, if a user requests a listing of all movies currently showing in San Francisco starring Jackie Chan, this query could be answered by going to a web site, such as the Internet Movie Database (imdb.com) to get a list of movies starring Jackie Chan and another site, (such as www.sfgate.com/eguide/movies/playing) to find out which of those movies are showing in San Francisco. In this case, the data sources are HTML documents and the information concerns movies, actors and theaters. An essential feature of this and other information-integration problems is that the interesting contents of the data sources can be completely characterized in terms of a simple logical language, such as SQL or Datalog. Although there are aspects of the HTML documents for these sites that are not captured by the logical description, they are not relevant to the problem of finding movie listings.

> *In information integration tasks, the end product is always information; once information is extracted from data, the data can be discarded, since all subsequent operations are on a logical representation of the information.*

Data manipulation concerns tasks like image processing and data swabbing (converting from one data format to another). In these tasks, data files are processed by one or more programs (filters) to produce new data files. In general, it is possible to describe the information

contained in the data in some logical language, but these descriptions do not completely characterize the data.

> *In data manipulation tasks, the end product is generally data; information may or may not be extracted from data files, and most operations act directly on the data.*

# 3 Two kinds of sensors

> ***data**: information output by a sensing device or organ that includes both useful and irrelevant or redundant information and must be processed to be meaningful*
>
> — Merriam-Webster OnLine (http://m-w.com)

Since data come from sensors, a good place to start the discussion of representing data is how to represent sensors. There are two types of sensors mentioned in the definition above. One is a "sensing device," such as a Geiger counter or other scientific instrument, which produces data that must in turn be sensed and interpreted. The other is a sense organ, such as an eye, which feeds information directly into the brain.

Planners for information integration represent sensors as sense organs. For example, the Internet Softbot [?] treats Unix commands like `ls`, which lists the files in a directory, as sense organs. To represent that executing the action `ls /bin` reveals the name of every file in `/bin`, the Softbot encoding (in the SADL language [?]) is:

$$\forall f \, \exists n \; \textbf{when} \; (\mathsf{in.dir}(f, \; \texttt{/bin})) \; \textbf{observe}(\mathsf{name}(f, \; n))$$

which translates to "The softbot will observe the name of every file in the directory `/bin`." The **observe** annotation means that the appropriate propositions of the form name($f$, $n$) will be inserted into the agent's knowledge base when the action is executed. This is the most direct and intuitive way to represent a sensor. What this encoding actually represents, however, is not `ls`, but the combination of `ls` with a program (called a *wrapper*) to read in and interpret its output. The wrapper itself is a "black box" to the planner; the planner "knows" what information is contained in the output of `ls`, but knows nothing of how that information is encoded. Thus, this representation is not suitable for data processing domains, in which the data output itself is of interest.

A sensing device, in contrast, produces an output or result that depends on the state of the world. This output may be perceived and interpreted to obtain information about the world, but the interpretation depends on knowing how the output was produced. This kind of indirect sensing, or *testing*, is exemplified by Moore's litmus paper example [?]. For an excellent formal analysis of testing, see [?]. The main idea behind testing is to exploit actions with conditional effects to obtain information about properties of the world that are not directly perceptible (such as the acidity of a solution or the amount of radiation emitted by some source). This strategy only works if the outcome of the conditional effect (such as the color of the litmus paper or the frequency of clicks from the Geiger counter) is directly perceptible, or can in turn be detected via some other test. Thus, testing must

always bottom out in some sense organ. In the litmus paper example, it bottoms out when the photons reflected from the paper strike the retina, stimulating neurons in the visual cortex, producing the sensation of *redness* or *blueness*. The agent then reasons backward from its causal theory to determine what the redness or blueness says about the acidity of the solution.

The softbot analogue of the retina is working memory, so direct perception occurs when the output of a sensor is loaded into working memory. Directly perceptible properties are properties of the output that can be computed without resorting to any additional information about the state of the outside world. Anything else must be inferred from the known causal relationship between the state of the and the contents of the data.

For example, the effect of `ls` discussed above would be written as

$$\forall f\text{: file (parent.dir}(f) \text{ = /bin)} \rightarrow \text{contains-line(name}(f)\text{, } out\text{)}$$

This translates to "For each file in directory `/bin`, there is a line in the output that is equal to the name of the file." The $\rightarrow$ is used instead of **when** (which is used in the SADL example above) to indicate a conditional effect. In this example, the meaning of the two notations is the same, but $\rightarrow$ has broader applicability as is discussed below. The predicate contains-line ($s_1$, $s_2$) means that string $s_1$ appears in string $s_2$, delimited by newlines. The truth value of this predicate can be computed given only the strings $s_1$ and $s_2$, thus satisfying the requirements stated above for direct observability. The function parent.dir($f$), on the other hand, cannot be directly perceived, but once the softbot knows what strings appear in the output *out*, it can infer what files are contained in the directory `/bin`.

There is a qualitative difference between predicates and functions like contains-line, which are directly perceptible and those like parent.dir, which are not:

- The function parent.dir is a *fluent*. That is, its value depends on the state of the world. Knowing the value of $f$ is not enough to know the value of parent.dir($f$); it can be different in different hypothetical states of the world and can change from time to time.

- The relation contains-line is *static*. Its value depends only on the values of its arguments, and no change to the world can change its value. Furthermore, its value can easily be computed. "Sensing" its value is reduced to loading its arguments into working memory and performing a simple computation. There are no preconditions or side-effects to such a computation, so an explicit sensing action is unnecessary; instead, we can associate with each static predicate or function a procedure for determining its value. Static predicates are sometimes called *facts*.

If we compare this approach to the approach used for information integration, it becomes clear that the computation performed in reading in the output of a data-producing action, executing procedures to determine the values of static predicates and inferring the values of fluents is exactly what a wrapper does. What we've done is changed the wrapper from a "black box" to a "white box," which is integrated into the reasoning process of the planner itself. This change in representation gives the planner the ability to reason about actions that manipulate data, something that is not possible given the "sense organ" representation of information-producing actions.

# 4 Representing metadata

The effect description of `ls` given above can be rewritten in the equivalent form[1]

> $\forall f$: file, $n$: filename (parent.dir($f$) = /bin) $\wedge$ $n$ = name($f$) $\rightarrow$ contains-line($n$, *out*)

This effect can be regarded as a *metadata* description of the output of `ls`. It contains two vital components:

1. the information contents of the data
2. how the information is encoded in the data

The $\rightarrow$ relates the two. The symbol $\rightarrow$ does not merely denote implication (which we will represent as "$\Rightarrow$"), since it involves a temporal element as well as a logical element. When it appears in the form of a conditional effect, the $\rightarrow$ relates the truth value of the expression on the left hand side (LHS) *before the action is executed* with the truth value of the expression on the right hand side (RHS) *after the action is executed.* However, since the expression on the RHS is static, the only time point of interest is that of the LHS. Thus, another component of metadata is:

3. what time the information pertains to

For data-producing actions, this time is whenever the action is executed, so there is no need to state it explicitly in the action description, but once the output has been produced, it is necessary to keep track of what time it pertains to. For example, if a softbot is to produce nightly backups, the night a given backup was made should be recorded.

For data *goals*, the time that the information pertains to is also of interest. As discussed in [?], failure to specify the time for which information is requested is the reason why a goal of "tell me the color of the door" could achieved by painting the door blue and then answering "blue." If instead we ask "what is the *current* color of the door," painting the door only obscures that information and does nothing to answer the question. The syntax of data goals is the same as that of data-producing effects, except that goals can refer explicitly to the time of interest for fluents on the LHS. This is done using an extra, optional argument for each fluent. For example, to refer to the color of the door at 9am today, we can write $c$ = color(door, 9:00)). If no time is specified, it is assumed to be whenever the goal is given. If a future time is specified, this is interpreted as a request to schedule a data-gathering operation to be carried out when that time arrives.

Another difference between data-producing actions and data goals is that the goals must specify what is to be done with the data, such as the pathname of the file where the data should be put. Note that pathname is a fluent, as it must be, since storing the data to a file changes the world. So another component of metadata is

[1] Throughout this paper, "P $\rightarrow$ f($g(x)$)" is a shorthand for "P $\wedge(y$ = g($x$)) $\rightarrow$ f($y$)" and "P $\rightarrow$ (f($x$) = g($x$))" is a shorthand for "P $\wedge(y$ = g($x$)) $\rightarrow$ (f($x$) = $y$)."

4. where the data reside

Although the location of the data is specified as a fluent, no time is given; for goals, it is assumed to be "as soon as possible" (as mentioned above, scheduling future data-gathering operations can be achieved by specifying a future time in the LHS of the goal); for metadata in the agent's knowledge base, it is assumed to be "right now," since anything else would amount to incorrect information about the world.

## 4.1 Limited completeness assumption

We make a limited form of the STRIPS assumption. We do not assume that all effects are explicitly listed in data-producing effects, since it is impossible to account for all possible effects. For example, concatenating together two files, neither of which is a valid GIF file could produce a result that *is* a valid GIF file. We do require, however, that if any predicate is used in the RHS of any effect or metadata description, then the description must be complete with respect to that predicate. For example, the output of `ls /bin` cannot contain any lines of text that are not the names of files in `/bin`. Any property that does not appear in the RHS of a metadata description is considered to be *unknown*.

## 4.2 Constraints

Many data processing domains require sophisticated constraint reasoning in order to select parameters appropriately [?, ?]. ADLIM provides "built-in" constraints, such as inequality. Additionally, all of the procedures for evaluating static predicates are implemented as constraints. That is, the set of possible values for each argument is restricted based on the other arguments. It is useful to know what arguments will be determined if singleton values are provided for other arguments. We use the predicate bound to represent that the value(s) of an argument are determined. For example, we can write

$$\text{contains-line}(s, out) \wedge \text{bound}(out) \rightarrow \text{bound}(s)$$

to represent that once the output *out* is known, every line of text in *out* can be determined. By definition, the return value of a function (or the boolean value of a relation) will be bound if all its arguments are bound. Additionally, by convention, the last argument of a static predicate is the data being described. If that argument is bound then all other arguments will be bound, as in the above example. This can, however, be overridden to capture *binding patterns*. Binding patterns are used to require particular arguments to be specified. For example, consider the predicate contains($s$, *out*), meaning *out* contains $s$ as a substring. It is *possible* to list all substrings of *out* once the value of *out* is known, but it would not be practical.

Additionally, unary constraints can be associated with types. For example, a Unix filename can be any non-empty string that does not contain the character "/." This constraint can be represented by the regular expression "~[/]+".

### 4.3 Example

Suppose `plot` is a grayscale image corresponding to an elevation map of the San Francisco Bay Area. Let xProj and yProj be linear functions mapping the x and y coordinates of the image to the corresponding longitude, latitude coordinates. Let hProj be a linear function mapping elevation to pixel values in the image, where lower (blacker) values correspond to lower elevations. Let elevation($x$, $y$, $w$, $h$) be a fluent function returning the average elevation over the $w$ by $h$ square centered at $x$, $y$. The metadata description of this plot would look like

$$\begin{aligned}
&\text{xSize}(\texttt{plot}) = \text{XMAX} \wedge \text{ySize}(\texttt{plot}) = \text{YMAX} \wedge \\
&\forall x, y\text{: natural, } el\text{: real} \\
&\quad x < \text{XMAX} \wedge y < \text{YMAX} \wedge el = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES}) \\
&\quad\quad \rightarrow \text{value}(x, y, \texttt{plot}) = \text{hProj}(el)
\end{aligned}$$

where words in ALL CAPS are constants. Note that whereas the description of `ls` quantified over lines of text in the output, this description quantifies over the pixels. It would almost never be desireable to extract the information associated with each pixel, but it is useful to describe the file in terms of what each pixel represents, since image-processing programs typically operate on pixels.

## 5 Filters

A *filter* is an action that transforms data. A filter has one or more inputs and one or more outputs, and the outputs depend on the inputs in some way, as specified in the action effect description. It does not modify its inputs in any way, and the outputs are always new objects. Furthermore, a filter has no side effects. As with data-producing actions, the effects of filters are specified using conditional effects. However, since filters don't produce information, the LHS cannot refer to the state of the world, but only to the input data. Since filters don't affect the world, the RHS also cannot refer to the state of the world, only output data. Thus, both the LHS and the RHS of effects are specified with static predicates.

For example, the Unix `grep` command outputs the lines of text appearing in its input that contain text matching a given regular expression. For example, "`grep .ps$`" outputs all strings from its input ending in ".ps":

$$\forall s\text{: string } \textsf{contains-line}(s, \textit{input}) \wedge \text{matches}(s, \text{".ps\$"}) \rightarrow \textsf{contains-line}(s, \textit{output})$$

Given this description, and the description of `ls`, it is easy to see that the output of "`ls /papers | grep .ps$`" (directing the output of `ls /bin` to the input of `grep .ps$`) will contain the names of all files in directory `/bin` that end in `.ps`.

## 5.1 Data mining

"Data mining" actions extract or call attention to information that is latent in the data but is not readily apparent. For example, a rock-finding algorithm might produce an output that lists or highlights probable rocks in an image. The information must have been in the data all along, but it was "hidden." We represent hidden properties of the data using static predicates that do not have constraints (extraction procedures) associated with them. For example, we might represent pixels corresponding to rocks in an image using the predicate rockPixel. A data mining action is then represented as a filter that maps from imperceptible static predicates like rockPixel to perceptible ones.

## 5.2 Example

Figure 1 shows a dataflow plan that includes the program `compose`, to compose red, green and blue monochrome images into a color image. Here we show a similar program, `HSVcompose`, which does the same thing, but using an HSV (hue, saturation, value) representation of the color. The inputs are three *valuemaps*, which are essentially monochrome images containing the hues, saturations and values that are to be combined. The output is a *colormap*, whose pixel values are numbers that can be also represented as RGB or HSV triples. The function HSVcolor returns the corresponding color value for a given h,s,v triple.

`action` HSVcompose()
`input` *hue, sat, val*: valuemap
`output` *HSVout*: colormap
`precond` xSize(*hue*) = xSize(*sat*) ∧ xSize(*hue*) = xSize(*val*) ∧
ySize(*hue*) = ySize(*sat*) ∧ ySize(*hue*) = ySize(*val*)
`forall` *x, y*: naturalNumber, *h, s, v*: pixelValue
`effect` xSize(*HSVout*) = xSize(*hue*) ∧ ySize(*HSVout*) = ySize(*hue*) ∧
( *x* < xSize(*hue*) ∧ *y* < ySize(*hue*) ∧ value(*x, y, hue*) = *h* ∧
value(*x, y, sat*) = *s* ∧ value(*x, y, val*) = *v*
→ color(*x, y, HSVout*) = HSVcolor(*h, s, v*) )

# 6 Other actions

## 6.1 Data delivery

Recall that a component of metadata is the specification of where the data reside. It is not sufficient to produce the data product; the agent must also do something with it, such as store it in a file, print it, or deliver it as email. A data delivery action is one that takes one or more inputs and changes the state of the world to produce some physical embodiment of the data. Again, we use conditional effects. The LHS refers to the data, and thus is expressed using static predicates. The RHS refers to the state of the world, and thus is expressed using fluents. The delivery of data also involves the creation of new objects in the world, such as files and printouts. We indicate the creation of a new object using the keyword **new**. For example, the action of saving a file with the pathname "`data.out`" would have an effect like

**new** $f$: file contents($f$) $\leftarrow$ *input* $\wedge$ pathname($f$) `data.out`

There is no LHS, since the action unconditionally saves its input.

### 6.2 Data construction

Not all data-producing actions return information about the world. Here's an action that just creates a value map of a specified size and fills it with a specified value.

**action** makeConstant($c$: pixelValue, *width*, *height*: naturalNumber)

`output` *MCout*: valueMap
`forall` $x$, $y$: naturalNumber
`effect` xSize(*MCout*) = *width* $\wedge$ ySize(*MCout*) = *height* $\wedge$
($x <$ *width* $\wedge y <$ *height* $\rightarrow$ value($x$, $y$, *MCout*) = $c$)

### 6.3 Causal actions

It is also possible to specify actions that unconditionally change the world, just as in other actions languages. Such actions can have no inputs or outputs.

## 7 Information gathering

Most data-processing plans pass data directly from one action to another, without any requirement for the agent to "know" the contents. However, there are cases in which the agent must explicitly gather information in support of planning, to determine the value of a parameter to an action or to make a decision.

In SADL, it is straightforward for an agent to determine what it "knows" as a result of executing actions, since these facts are expressed directly as **observe** effects and are inserted into the agent's knowledge base. In ADLIM, information gathering requires two steps: (1) finding or constructing a data file that contains the desired information and (2) extracting the information from the file.

### 7.1 The information contained in data

We first turn to the question of what information is contained in data described by a given metadata expression. We have described metadata as a mapping from information to the way that information is encoded in the data. To extract information from data, it is necessary to run that mapping backward: given the bits and bytes of a data file, determine what information is contained therein. The limited completeness assumption discussed above this reverse mapping possible. For example, in the case of the output of `ls /bin`, we assume that every occurance of contains-line is reflected by some metadata explanation, and since the only explanation of contains-line provided by `ls` is that the string is the name of a file in `/bin`, it must be the case that every line of the output is the name of a file in `/bin`.

Intuitively, a data source (such as `ls /bin`) provides information about particular attributes (e.g., name) of members of some set (e.g., files in `/bin`). The set membership is defined by the entire LHS expression. For example, `ls /bin` lists the names of all files in `/bin` that have names (which happens to be all of them). Similarly, since only objects satisfying the LHS will be listed, everything in the LHS will be known to be true of objects that are listed. For example, it will be known that every file listed in the output is in the directory `/bin`, even though `/bin` itself does not appear in the output.

However, it is still possible for an attribute to restrict the set membership without having its value known for members of the set. For example, consider a program that lists the pathname of every file larger than a gigabyte (GB) to the output *out*:

$$\forall f, n, s\ (\mathsf{pathname}(f) = n) \wedge (\mathsf{size}(f) = s) \wedge (s > 10^9) \rightarrow \mathsf{contains\text{-}line}(n, out)$$

Here, size is used to restrict the membership, but the size of the files is not listed in *out*. All that will be known about the size of files listed in *out* is that it is greater than one GB. As this example reveals, one way an attribute can be unknown is if a variable from that attribute appears in the LHS but not in the RHS. In this case, that variable is $s$. Thus, $\mathsf{size}(f)$, which is specified in terms of $s$, will remain unknown. There is one other way that information about attributes can be missing from the data: if the LHS is disjunctive (or equivalently, the same RHS appears in multiple metadata rules). For example, if we have the following expressions

$$\forall f\ \mathsf{pathname}(f) = n \wedge \mathsf{parent.dir}(f) = /\mathtt{bin} \rightarrow \mathsf{contains\text{-}line}(n, out)$$
$$\forall f\ \mathsf{pathname}(f) = n \wedge \mathsf{parent.dir}(f) = /\mathtt{baz} \rightarrow \mathsf{contains\text{-}line}(n, out)$$

then *out* will contain the names of files in `/bin` and `/baz`, so the only thing that will be known about a string appearing in *out* is that it is *either* the name of a file in `/bin` *or* the name of a file in `/baz`. ADLIM does not support such disjunctive world knowledge, so we will only consider cases in which the LHS is conjunctive.

## 7.2 Local completeness

Such information about all members of a set is called Local Closed-World (LCW) or Local Completeness (LC) information, which was introduced by [?, ?] and extended by [?] and [?]. We will describe the formulation of LC from the Razor system [?], since that is the most general of the three, and also the closest to our metadata representation.

A local completeness statement in [?] is expressed as a horn clause, where the head is specified using a *site relation*, which denotes the tuple of values returned by a given site or database query, and the tail is specified using *world relations*, which describe the tuples returned in terms of a site-independent ontology. World relations are essentially what we call fluents, and site relations are similar to our static predicates. For example, in [?].

$$\text{IMDBCAST(M, A)} \Leftarrow \mathsf{actor\text{-}in}(\text{M, R, A}) \wedge \mathsf{year\text{-}of}(\text{M,Y}) \wedge (\text{Y} \leq 1996)$$

is used to represent the statement that the Internet Movie Database (IMDB) lists the actors of all movies made before 1997). IMDBCAST is a site relation, describing what tuples the query to IMDB returns, and the other relations are world relations. Although the year, Y, is used to limit the scope of the LC statement, it is not returned by the query. This example is essentially the same as the example above of a command the returns all files above one GB in size.

LC can also be used to state that one site contains all the information contained in another, by including site relations in the body of the LC statement. This sort of metadata could also be expressed in ADLIM. In fact, that is essentially what the description of a filter is. For example, from the description of `grep`, we can conclude that the input subsumes the output.

ADLIM metadata expressions are strictly more expressive than LC. Any LC statement could be represented as a metadata rule by introducing a static predicate corresponding to the site predicate, where the procedure for evaluating it involves a database query. Some differerences are:

1. The site descriptions in [?] do not concern themselves with the representation of filters, since filters are unnecessary in pure information-integration domains.

2. Although site relations are somewhat analagous to static predicates in ADLIM, they are not the same thing. Static predicates are site-independent and capture the information implicit in a wrapper in a system like Razor. Site relations are site-dependent.

3. The data description in ADLIM rules can be arbitrarily complex, whereas LC statements only allow one to specify the tuples returned by a site.

# 8 Reasoning about plans

We have implemented a planner that supports a large subset of the Adlim language, and we are in the process of improving it, both in functionality and in efficiency. A discussion of the planner is beyond the scope of this paper, but here we briefly discuss how planning can be done using Adlim, and we show a concrete example.

A planning problem is a triple $\prec I, \Lambda, \Gamma \succ$, where $\mathcal{I}$ is a specification of the initial state, including the agent's metadata database, $\Lambda$ is a description of the actions available, in the form of a list of action schemas, and $\Gamma$ is a goal description, which may be a metadata specification of desired data. The purpose of planning is to generate a plan from the actions in $\Lambda$ that results in $\Gamma$ being satisfied if it is executed in a state satisfying $I$.

A plan is a triple $\prec A, O, B \succ$, where $A$ is a set of action schemas, $O$ is set of ordering constraints over $A$, and $B$ is a set of bindings on variables in $A$, including parameter assignments and assignments of outputs to inputs. It is the latter that makes plans serve as dataflow programs. All inputs must be bound to either the output of an earlier action or some data file for which there is a metadata description in $I$, and not input may be bound to multiple outputs.

Since a goal is a metadata expression, it has a LHS, which refers to the initial state (or earlier), and a RHS, which refers to the final state. Planning can be done using regression,

in which the RHS is regressed backward in time until the initial state plus the LHS entail the RHS. Or it can be done using progression, in which the current state and the LHS are progressed forward in time until they entail the RHS. Since the LHS specifies the information that is desired, it can provide substantial guidance to the search.

## 8.1 Example

We will show an example of planning by goal regression. Let's return to the elevation map of the San Francisco Bay, discussed previously. Suppose we want to produce a color image identical to this image, except that pixel values corresponding to points below sea level are blue — darker blue corresponding to greater depth. We can best describe this goal using an HSV (hue, saturation, value) representation of the color. All points should have the same value (brightness) as the original elevation map. Points above sea level should have zero saturation (gray pixels). Points below sea level should have a hue of blue and maximum saturation.

$\forall x, y$: naturalNumber, $h, s, v$: pixelValue, *elev*: real
$(x < \text{XMAX} \wedge y < \text{YMAX} \wedge \ elev \ = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES})) \rightarrow$
color($x$, $y$, *map*) = HSVcolor($h$, $s$, hProj(*elev*)) $\wedge$ $(elev \ > 0 \rightarrow s = 0) \wedge$
$(elev \ \leq 0 \rightarrow s = \text{MAXVALUE} \wedge h = \text{BLUE})$

The goal does not specify the hue for pixels corresponding to points above sea level, since the hue is irrelevant if the saturation is zero. This goal can be solved by using `HSVcompose`, where the value map is the elevation map, the hue map is a solid BLUE, and the saturation map is the result of thresholding the elevation map, such that values below zero elevation correspond to MAXVALUE pixels and values above correspond to zero. If we regress the goal through `HSVcompose`, with the I/O assignment *map* = *HSVout*, we get a new goal, in which color($x$, $y$, *map*) = HSVcolor($h$, $s$, hProj(*elev*)) is deleted (since it is satisfied by `HSVcolor`), and the preconditions of `HSVcolor` (underlined) are added.

$x < \text{XMAX} \wedge y < \text{YMAX} \wedge \ elev \ = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES}) \rightarrow$
$\underline{\text{xSize}(hue) = \text{xSize}(sat) = \text{xSize}(val) \wedge \text{ySize}(hue) = \text{ySize}(sat) = \text{ySize}(val)} \ \wedge$
$\underline{x < \text{xSize}(hue) \wedge y < \text{ySize}(hue) \wedge}$
$\underline{\text{value}(x, y, hue) = h \wedge \text{value}(x, y, sat) = s \wedge \text{value}(x, y, val) = \text{hProj}(elev) \wedge}$
$(elev \ > 0 \rightarrow s = 0) \wedge \ (elev \ \leq 0 \rightarrow s = \text{MAXVALUE} \wedge h = \text{BLUE})$

We need an action to produce a threshold map corresponding to sea level. The arguments are the threshold value and the values to assign to pixels that that fall below and above the threshold.

`action` threshold(*thresh*, *below*, *above*: pixelValue)
`input` *THin*: valueMap
`output` *THout*: valueMap
`forall` $x$, $y$: naturalNumber, $v$: pixelValue
`effect` xSize(*THout*) = xSize(*THin*) $\wedge$ ySize(*THout*) = ySize(*THin*) $\wedge$

$$(x < \text{xSize}(THin) \wedge y < \text{ySize}(THin) \wedge v = \text{value}(x, y, THin) \rightarrow (v \leq thresh \rightarrow \text{value}(x, y, THout) = below) \wedge (v > thresh \rightarrow \text{value}(x, y, THout) = above))$$

We can then regress the above goal through `threshold`(hProj(0), MAXVALUE, 0), with I/O assignment $sat = THout$. xSize($sat$), ySize($sat$) and value($x$, $y$, $sat$) are deleted, and the non-redundant preconditions of `threshold` (underlined) are added.

$$x < \text{XMAX} \wedge y < \text{YMAX} \wedge elev = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES}) \rightarrow \text{xSize}(hue) = \underline{\text{xSize}(THin)} = \text{xSize}(val) \wedge \text{ySize}(hue) = \underline{\text{ySize}(THin)} = \text{ySize}(val) \wedge x < \text{xSize}(hue) \wedge y < \text{ySize}(hue) \wedge \underline{v' = \text{value}(x, y, THin)} \wedge \text{value}(x, y, hue) = h \wedge \text{value}(x, y, val) = \text{hProj}(elev) \wedge (elev > 0 \rightarrow \underline{v' \geq \text{hProj}(0)}) \wedge (elev \leq 0 \rightarrow \underline{v' \leq \text{hProj}(0)} \wedge h = \text{BLUE})$$

Regressing through `makeConstant`(BLUE, XMAX, YMAX) with the I/O assignment $hue = MCout$, value($x$, $y$, $hue$) = $h$, $x <$ xSize($hue$), $y <$ ySize($hue$) and $h$ = BLUE are satisfied and we get

$$x < \text{XMAX} \wedge y < \text{YMAX} \wedge elev = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES}) \rightarrow \underline{\text{XMAX}} = \text{xSize}(THin) = \text{xSize}(val) \wedge \underline{\text{YMAX}} = \text{ySize}(THin) = \text{ySize}(val) \wedge \text{value}(x, y, val) = \text{hProj}(elev) \wedge \underline{x < \text{XMAX} \wedge y < \text{YMAX}} \wedge v' = \text{value}(x, y, THin) \wedge (elev > 0 \rightarrow v' > \text{hProj}(0)) \wedge (elev \leq 0 \rightarrow v' \leq \text{hProj}(0))$$

Matching against the initial state with the I/O assignment $THin$ = `plot` and $val$ = `plot`,

$$x < \text{XMAX} \wedge y < \text{YMAX} \wedge elev = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES}) \rightarrow x < \text{XMAX} \wedge y < \text{YMAX} \wedge \underline{elev = \text{elevation}(\text{xProj}(x), \text{yProj}(y), \text{XRES}, \text{YRES})} \wedge elev > 0 \rightarrow \underline{\text{hProj}(elev)} > \text{hProj}(0) \wedge elev \leq 0 \rightarrow \underline{\text{hProj}(elev)} \leq \text{hProj}(0)$$

The first three terms on the RHS are entailed by the LHS. The rest follows from the fact that hProj is an increasing linear function.

# 9 Conclusions

There are a surprising number of interesting representational problems that arise when representing and reasoning about metadata. Constraint reasoning is complicated by the fact that some expressions contain variables whose values are unknown and universal quantification over unknown universes. Constraint reasoning systems typically deal with variables of known, fixed domains. "Philisophical" problems like the Identity Problem come up regularly in this domain (and indeed many softbot domains). These problems are far from insurmountable, but they do require some thought on the part of the domain designer.